# Fork-join parallelism

5/6/26

(drawing on materials by Dan Grossman)

# Administrivia

- HW 6 (BSTs, tree traversals, design problems) due Friday
- Exam 2 is on Monday
  - In class, open note, written exam
  - Topics:
    - Data structures: Stack, Queue, Set, Map, Priority Queue, StringBuilder
    - Sorting and comparators
    - Design problems
    - Binary search and binary search trees
    - Tree traversals
    - More questions on running time
    - Search (backtracking and connected components)
    - Inheritance

# Using multiple cores

- Essentially all computers now are multi-core (dual-core means 2 cores, quad-core means 4 cores, etc)
  - Each core can run all the operations needed to run a program
- Up to this point, our programs have always done a single thing at once so one can run on each core
- What if we wanted multiple cores to run a single program faster?
  - Need to split program into parts that can run on different cores

# Sample problem: Summing an array

```
int simpleSum(int[] array, int low, int hi) {
        int sum = 0;
        for(int i=low; i < hi; i++)
                sum = sum + array[i];
        return sum;
}
```

# Splitting the sum into two parts

```
int sum(int[] array, int low, int hi) {
        int left = simpleSum(array, low, (hi+low)/2);
        int right = simpleSum(array, (hi+low)/2, hi);
        return left + right;
}
```

# New conceptual keywords

(not real Java)

```
int sum(int[] array, int low, int hi) {
        int left = fork simpleSum(array, low, (hi+low)/2);
        int right = simpleSum(array, (hi+low)/2, hi);
        join;
        return left + right;
}
```

Allows child call (rest of line) to run in parallel with its parent

Wait for **all** children to complete

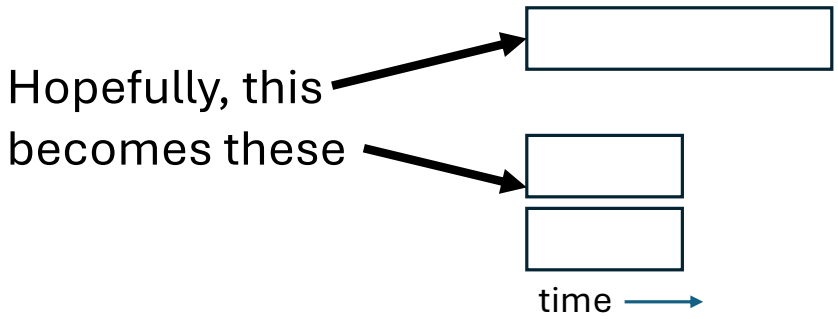
# New conceptual keywords

(not real Java)

```
int sum(int[] array, int low, int hi) {
        int left = fork simpleSum(array, low, (hi+low)/2);
        int right = simpleSum(array, (hi+low)/2, hi);
        join;
        return left + right;
}
```

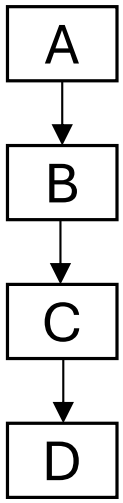
Allows child call (rest of line) to run in parallel with its parent

Wait for **all** children to complete



# fork and join in pictures

```
A();
B();
C();
D();
```



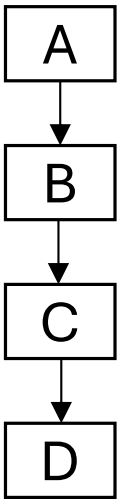

Each line must run in the given order

Boxes are *strands* (chunk of code we want to run without a fork or join)

Arrows show *precedence constraints*, relationship where one strand must run before another

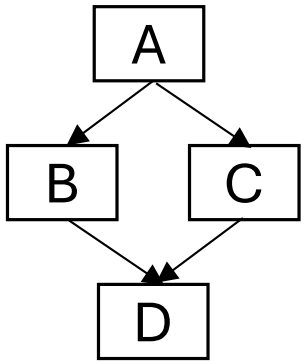
# fork and join in pictures

```
A();
B();
C();
D();
```




Each line must run in the given order

```
A();
fork B();
C();
join;
D();
```




Both B and C must run after A and before D, but their relationship is undefined

# Which of the following could be printed by the code below?

```
fork System.out.print(1);
System.out.print(2);
fork System.out.print(3);
join;
System.out.print(4);
```

I. 3124
II. 2134
III. 1243

A. Only I.
B. Only II.
C. Only I. and II.
D. Only II. and III.
E. None of the above

# Which of the following could be printed by the code below?

```
fork System.out.print(1);
System.out.print(2);
fork System.out.print(3);
join;
System.out.print(4);
```

I. 3124
II. 2134
III. 1243

A. Only I.
B. Only II.
C. Only I. and II.
D. Only II. and III.
E. None of the above

# Which of the following could be printed by the code below?

```
System.out.print(1);
fork System.out.print(2);
System.out.print(3);
System.out.print(4);
join;
```

I. 1234
II. 2134
III. 1324

A. Only I.
B. Only II.
C. Only I. and II.
D. Only II. and III.
E. None of the above

# Which of the following could be printed by the code below?

```
System.out.print(1);
fork System.out.print(2);
System.out.print(3);
System.out.print(4);
join;
```

I. 1234
II. 2134
III. 1324

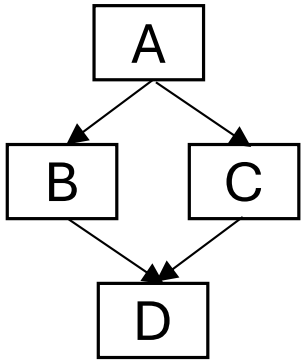
A. Only I.
B. Only II.
C. Only I. and II.
D. Only II. and III.
E. None of the above (I. and III.)

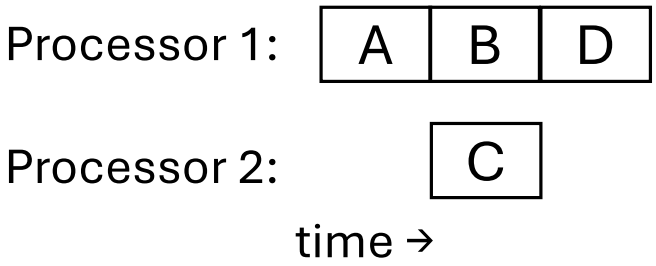
# Running a precedence graph

- Have a pool of cores/processors/threads (“processors”) that are available to run strands
- Each time unit, strands w/o unfinished predecessors are assigned to available processors and completed
- Typically assume strands represent equal amounts of work
- Processors are never voluntarily idle

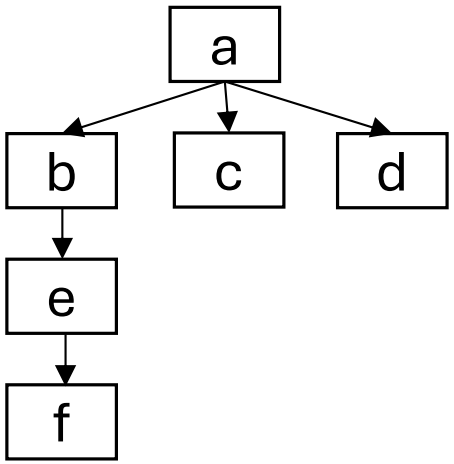
# Running a precedence graph

- Have a pool of cores/processors/threads (“processors”) that are available to run strands
- Each time unit, strands w/o unfinished predecessors are assigned to available processors and completed
- Typically assume strands represent equal amounts of work
- Processors are never voluntarily idle



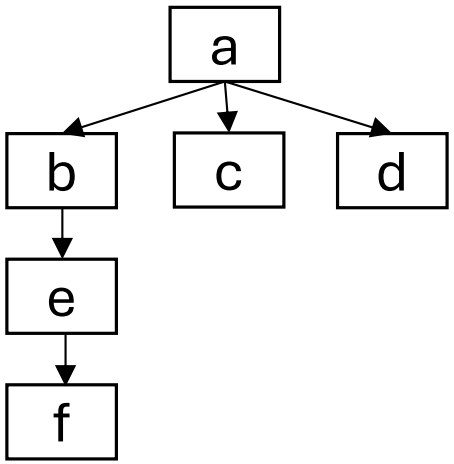



# How long will this graph run on 2 processors?



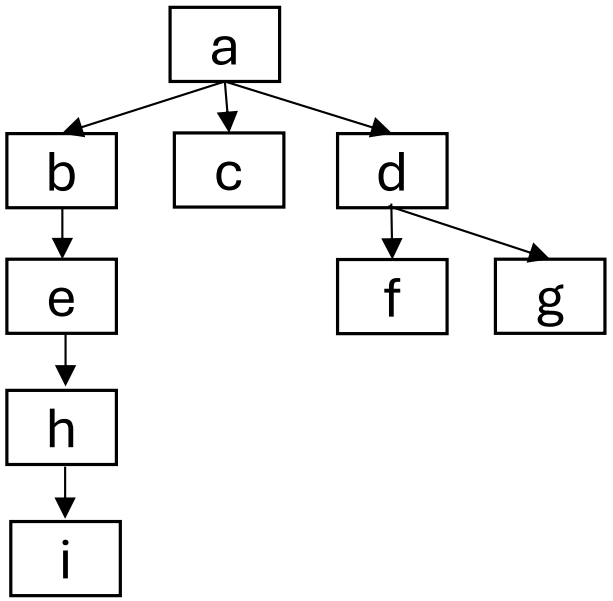

A. Exactly 3 time units
B. Exactly 4 time units
C. Either 3 or 4 time units
D. Either 4 or 5 time units
E. None of the above

# How long will this graph run on 2 processors?



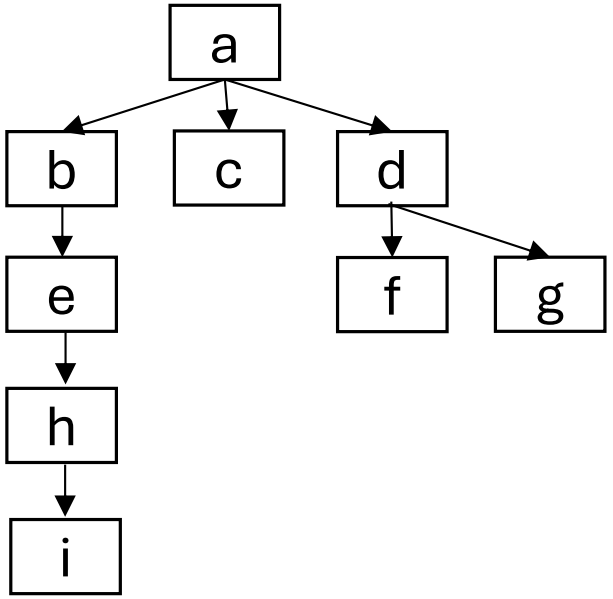

A. Exactly 3 time units
B. Exactly 4 time units
C. Either 3 or 4 time units
D. Either 4 or 5 time units
E. None of the above

# How long will this graph run on 2 processors?




A. Exactly 4 time units
B. Exactly 5 time units
C. Either 4 or 5 time units
D. Either 5 or 6 time units
E. None of the above

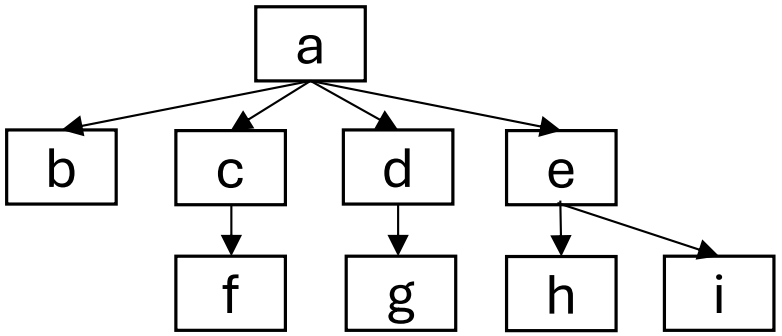
# How long will this graph run on 2 processors?




A. Exactly 4 time units
B. Exactly 5 time units
C. Either 4 or 5 time units
D. Either 5 or 6 time units
E. None of the above

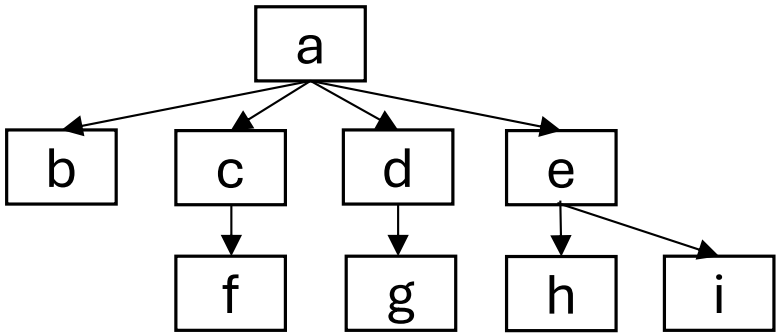
(between 5 and 7)

# How long will this graph run on 2 processors?




A. Exactly 4 time units
B. Exactly 5 time units
C. Either 4 or 5 time units
D. Either 5 or 6 time units
E. None of the above

# How long will this graph run on 2 processors?




A. Exactly 4 time units
B. Exactly 5 time units
C. Either 4 or 5 time units
D. Either 5 or 6 time units
E. None of the above

# Now we're trying to optimize 2 things

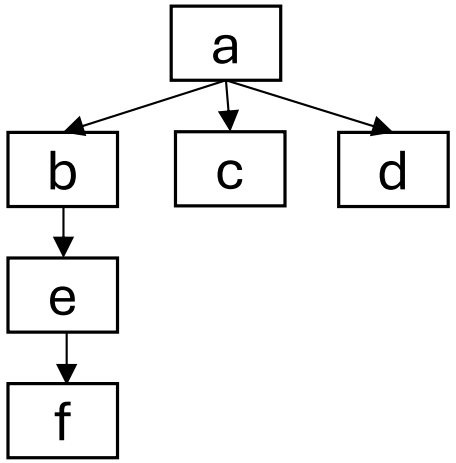
- Work = number of strands (denoted $T_1$)
- Span = length of longest path (denoted $T_\infty$)

# Now we're trying to optimize 2 things

- Work = number of strands (denoted $T_1$)
- Span = length of longest path (denoted $T_\infty$)

What are the work and span of this graph?

A. 3 and 4
B. 4 and 4
C. 6 and 3
D. 7 and 3
E. None of the above

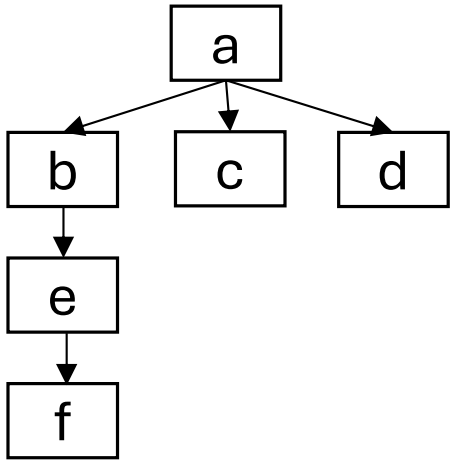


# Now we're trying to optimize 2 things

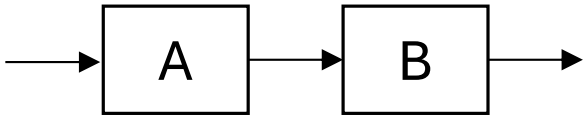
- Work = number of strands (denoted $T_1$)
- Span = length of longest path (denoted $T_\infty$)

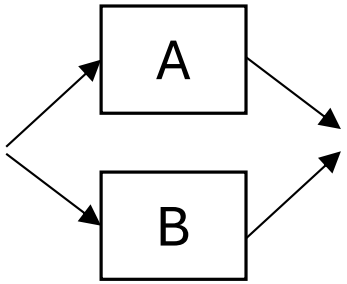
What are the work and span of this graph?

A. 3 and 4
B. 4 and 4
C. 6 and 3
D. 7 and 3
E. None of the above (6 and 4)

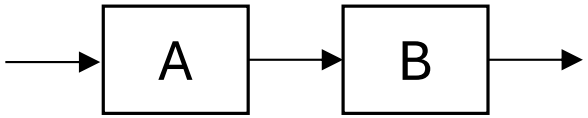


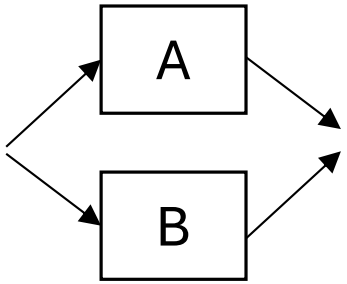
# Combining subcomputations



# Combining subcomputations

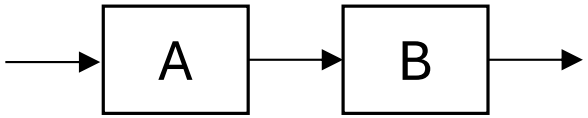


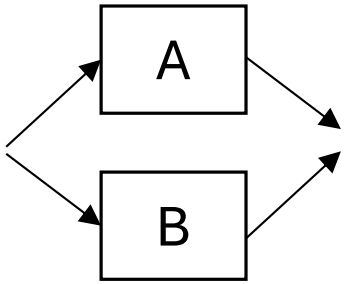




Work: $T_1(A) + T_1(B)$

Work: $T_1(A) + T_1(B)$

# Combining subcomputations




Work: $T_1(A) + T_1(B)$

Span: $T_\infty(A) + T_\infty(B)$

Work: $T_1(A) + T_1(B)$

Span: $\max\{ T_\infty(A), T_\infty(B) \}$

# Using different (fixed) numbers of cores

```
int sum(int[] arr, int low, int hi) {
    int p1 = (hi+2*low)/3;
    int p2 = (2*hi+low)/3;
    int s1 = fork simpleSum(array, low, p1);
    int s2 = fork simpleSum(array, p1, p2);
    int s3 = simpleSum(array, p2, hi);
    join;
    return s1 + s2 + s3;
}
```

```
int sum(int[] arr, int low, int hi) {
    int p1 = (hi+3*low)/4;
    int p2 = (2*hi+2*low)/4;
    int p3 = (3*hi+low)/4;
    int s1 = fork simpleSum(array, low, p1);
    int s2 = fork simpleSum(array, p1, p2);
    int s3 = fork simpleSum(array, p2, p3);
    int s4 = simpleSum(array, p3, hi);
    join;
    return s1 + s2 + s3 + s4;
}
```

# Generalizing to any number of cores

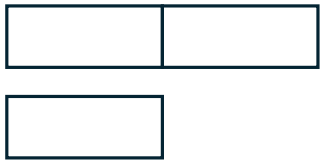
```
int sum(int[] arr, int low, int hi) {
    int c = getNumCores();
    int[] sums = new int[c];
    for(int i=0; i < c; i++) {
        int s = (i*hi + (c-i)*low) / c;
        int e = ((i+1)*hi + (c-i-1)*low) / c;
        sums[i] = fork simpleSum(array, s, e);
    }
    join;
    return simpleSum(sums, 0, c);
}
```

# Generalizing to any number of cores

```
int sum(int[] arr, int low, int hi) {
    int c = getNumCores();
    int[] sums = new int[c];
    for(int i=0; i < c; i++) {
        int s = (i*hi + (c-i)*low) / c;
        int e = ((i+1)*hi + (c-i-1)*low) / c;
        sums[i] = fork simpleSum(array, s, e);
    }
    join;
    return simpleSum(sums, 0, c);
}
```

What if

… c is wrong? (the OS takes a core back)

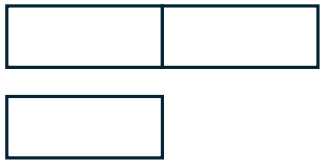


# Generalizing to any number of cores

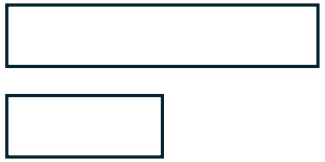
```
int sum(int[] arr, int low, int hi) {
    int c = getNumCores();
    int[] sums = new int[c];
    for(int i=0; i < c; i++) {
        int s = (i*hi + (c-i)*low) / c;
        int e = ((i+1)*hi + (c-i-1)*low) / c;
        sums[i] = fork simpleSum(array, s, e);
    }
    join;
    return simpleSum(sums, 0, c);
}
```

What if

... c is wrong? (the OS takes a core back)



... strands need different amounts of work?



# Recursion to the rescue!

```
int sum(int[] array, int low, int hi) {
        //base case
        int left = fork sum(array, low, (hi+low)/2);
        int right = sum(array, (hi+low)/2, hi);
        join;
        return left + right;
}
```

# Recursion to the rescue!

```
int sum(int[] array, int low, int hi) {
        //base case
        int left = fork sum(array, low, (hi+low)/2);
        int right = sum(array, (hi+low)/2, hi);
        join;
        return left + right;
}
```

Creates many more strands than cores

As each core finishes a strand, it gets another

Strands may have initial assignment, but any core can potentially take them (*work stealing*)